Capítulo 7

Clases

7.1. Introducción

7.1.1. Programación orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código alrededor de objetos, los cuales son instancias de clases. En lugar de enfocarse en las acciones y funciones que se realizan en el programa, la POO se centra en la interacción entre los objetos, permitiendo modelar y representar de manera más precisa entidades del mundo real en el software.

En la POO, un objeto combina datos y funciones relacionadas en una sola estructura, lo que facilita su manipulación y comprensión. Los objetos se definen mediante clases, que actúan como plantillas o moldes para crear instancias específicas. Cada objeto tiene atributos (propiedades o características) que describen su estado y métodos (funciones o comportamientos) que definen su capacidad para realizar acciones y responder a eventos.

El enfoque orientado a objetos fomenta la reutilización de código y el diseño modular, ya que los objetos pueden ser utilizados en diferentes partes de un programa y en otros programas. Además, permite una abstracción más efectiva, ya que los objetos encapsulan tanto los datos como el comportamiento relacionado, lo que facilita el desarrollo y el mantenimiento del software a largo plazo.

La POO se basa en cuatro principios fundamentales:

Abstracción Permite representar características y comportamientos esenciales de un objeto en un modelo simplificado. Se identifican las propiedades (atributos) y acciones (métodos) relevantes de un objeto y se encapsulan en una clase.

Encapsulamiento Consiste en ocultar los detalles internos de un objeto y proporcionar una interfaz para interactuar con él. Los datos (atributos) y las funciones (métodos) de un objeto se encapsulan dentro de la clase, y solo se accede a ellos a través de métodos públicos definidos en la interfaz.

Herencia Permite crear nuevas clases basadas en clases existentes, aprovechando las características y comportamientos heredados. Una clase derivada (subclase) puede extender o especializar la funcionalidad de una clase base (superclase), heredando sus atributos y métodos, y agregando o modificando los necesarios.

Polimorfismo Permite que un objeto pueda presentar diferentes formas o comportamientos según el contexto en el que se utilice. Se puede invocar un método de diferentes objetos de una jerarquía de clases y obtener resultados diferentes, siempre que cumplan con la interfaz común definida en una clase base.

La programación orientada a objetos (POO) fue inventada con el objetivo de proporcionar una mejor forma de organizar y diseñar software complejo. Antes de la POO, los programas se estructuraban principalmente en torno a funciones y procedimientos, lo que a menudo resultaba en un código difícil de mantener, extender y reutilizar.

La POO surgió en la década de 1960 como una forma de abordar estos desafíos. Su objetivo principal era permitir una representación más fiel de los objetos del mundo real dentro del software, lo que facilitaría el desarrollo de aplicaciones más escalables y flexibles.

Al enfocarse en los objetos, la POO permitió una mejor modelización de los sistemas, ya que los objetos podían contener tanto datos (atributos) como comportamientos relacionados (métodos). Esto permitió que el código se estructurara de manera más intuitiva y modular, ya que los objetos podían interactuar entre sí mediante mensajes, encapsulando así su funcionalidad interna.

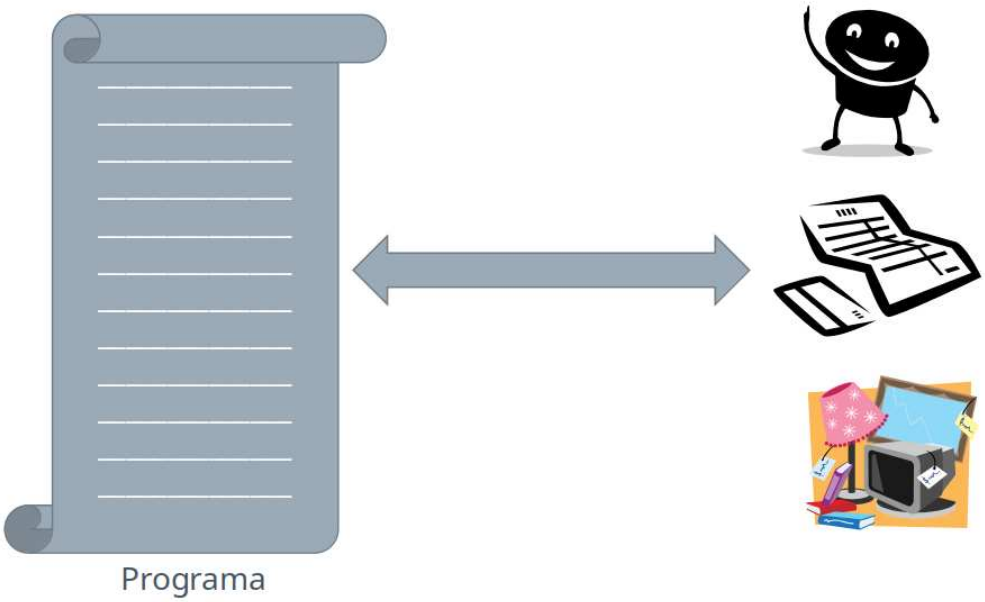


Figura 7.1: Programación estructurada

La programación orientada a objetos (POO) ofrece varias ventajas en comparación con la programación estructurada, especialmente en relación al concepto de la brecha semántica o semantic gap. La brecha semántica se refiere a la diferencia entre cómo se modela y se representa el mundo real en el software y cómo se percibe y comprende el mundo real por parte de los usuarios y desarrolladores.

De esta manera, si se requiere desarrollar un software que represente, por ejemplo, un sistema de ventas para un comercio, entre los conceptos principales detectados durante la etapa de análisis se encontrarían los de artículos, clientes y ventas. En la programación estructurada, el software sería redactado como una secuencia bastante larga de líneas de código, agrupadas y organizadas con algún criterio, pero orientado principalmente a la funcionalidad esperada por el software.

Sin embargo, todo software requiere ser susceptible a cambios luego de ser desarrollado. Tales cambios pueden provenir de diversos orígenes, tales como errores detectados durante el funcionamiento, requerimientos nuevos o incluso cambios externos, por ejemplo en la legislación vigente.

La programación orientada a objetos busca que el código fuente sea lo más parecido posible a la realidad, para que sea más simple identificar el punto donde debe ser modificado cuando se encuentre una modificación en los requerimientos, y que estos afecten en pocos lugares dentro de todo el código fuente. Así, si en el software del sistema de ventas, si hay un cambio en un requirimiento del proceso de ventas, los cambios en el código fuente estarían localizados principalmente en la clase que representa a la transacción de la venta, y no en la o las clases que representan a los artículos o a los clientes.

7.1.2. Clases y objetos

Cuando se busca resolver un problema utilizando la programación orientada a objetos, se aborda la representación del problema mediante la creación de objetos y las relaciones entre ellos. Cada objeto encapsula su propio estado y comportamiento, lo que permite modelar de manera más precisa las entidades y relaciones del dominio del problema.

Cada objeto tiene atributos que describen sus características y métodos que definen sus comportamientos o acciones asociadas. Muchos objetos comparten características que los definen, pudiendo ser agrupados o clasificados bajo una misma definición. Por ejemplo, dos objetos que realicen compras en un comercio generalmente tienen el mismo conjunto de datos y realizan las mismas acciones. A ambos se los puede clasificar como “Clientes”.

Una clase es una plantilla que define las características comunes compartidas por un conjunto de objetos para los que se aplica la misma definición. En esencia, una clase actúa como un modelo o plano para la creación de objetos. Contiene la estructura y el comportamiento que se aplicarán a cada instancia de esa clase.

Cuando se crea un objeto en la programación orientada a objetos, se basa en una clase específica. Cada objeto es una instancia individual y concreta de esa clase. Esto significa que cada objeto posee sus propios valores únicos para los atributos definidos en la clase, así como su propia capacidad para ejecutar los métodos asociados a esa

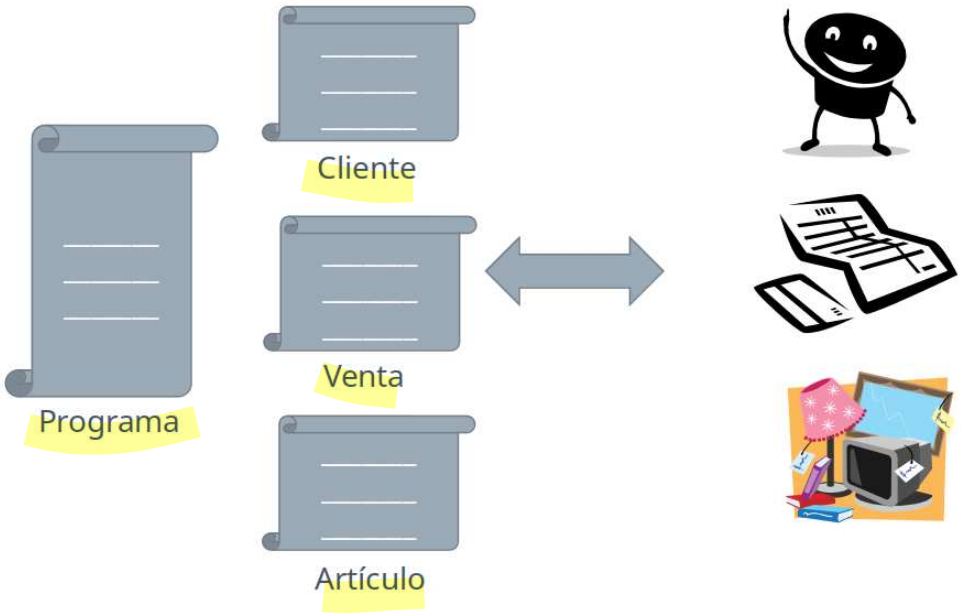


Figura 7.2: Programación orientada a objetos

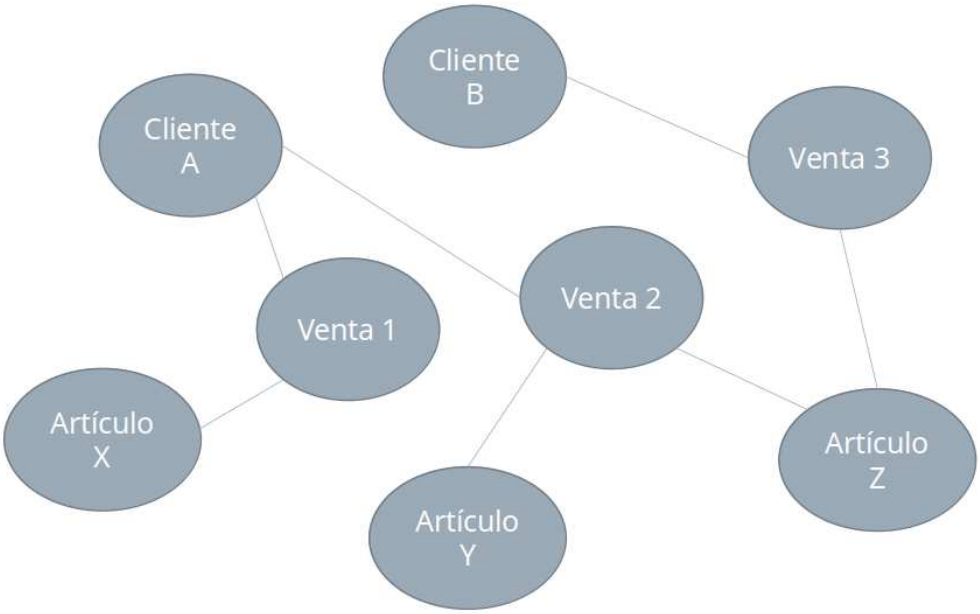


Figura 7.3: Objetos y sus relaciones

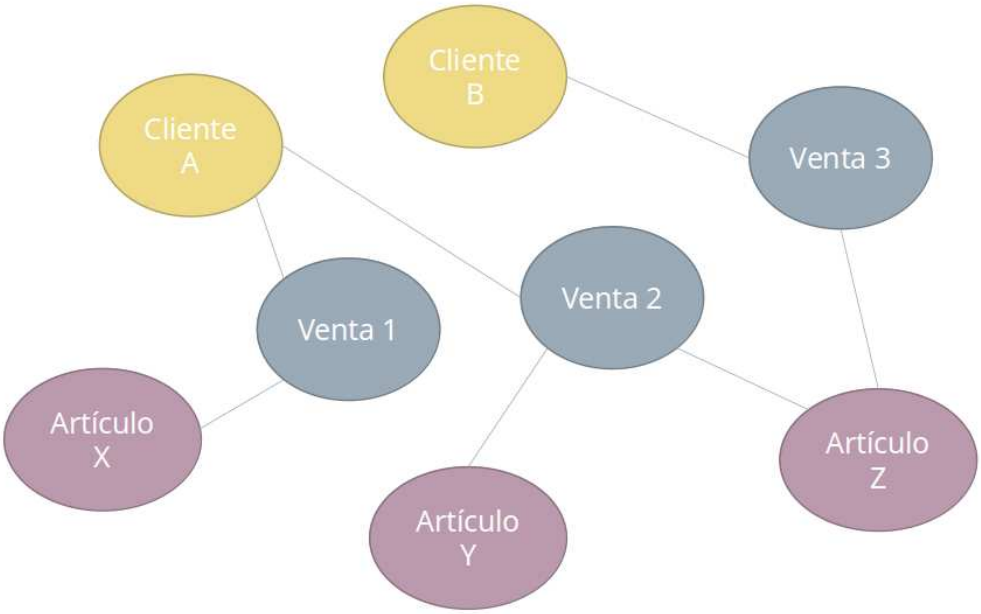


Figura 7.4: Objetos clasificados según su rol en el dominio

clase. Cada objeto es una entidad única y distinta que comparte las características y el comportamiento definidos en su clase, pero puede tener valores de atributos diferentes de las otras instancias de la misma clase y puede responder de manera independiente a los mensajes que reciba.

7.2. Clases

Para la creación de una clase se utiliza el bloque `class`, cuya sintaxis es:

```
class NombreClase:
    definición de métodos
```

El nombre de la clase generalmente consiste de un sustantivo en singular, con notación Pascal, es decir que esta commpuesto por letras en minúscula con su inicial en mayúscula. En caso de que el identificador esté formado por más de una palabra, se las diferencia sin un delimitador y con mayúscula en la primera letra de cada palabra, por ejemplo: CodigoPostal.

7.3. Atributos

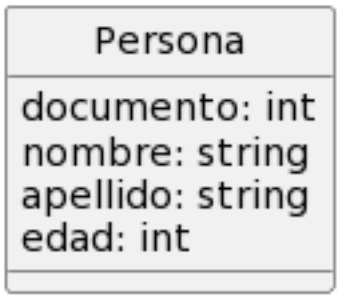
Los atributos no son declarados con una construcción gramatical en particular, simplemente se los asigna en el momento en que se los requiere crear dentro de algún método. Para diferenciar los atributos de las variables locales se los diferencia usando siempre la sintaxis `objeto.atributo`, indicando al objeto con el identificador seleccionado para referenciar al receptor del mensaje.

7.4. Constructor

El método constructor se denomina `__init__` y para funcionar require recibir como primer parámetro formal una variable que referencia al objeto que se está construyendo. El identificador de este parámetro puede ser cualquier nombre válido de variable, pero la convención prefiere denominarlo usualmente `self`.

El uso más habitual del método constructor es el de asignar valores iniciales para los atributos. Tales valores pueden ser recibidos como parámetros o asignados en forma arbitraria. Si bien las clases sólo pueden poseer un único constructor, se le pueden asignar valores por defecto a los parámetros para ofrecer que en la construcción se indiquen valores concretos durante la creación o se utilicen los valores por defecto.

Suponiendo una clase Persona con atributos para almacenar documento, nombre, apellido y edad de una persona, se presentan a continuación el diagrama de clases y el bloque class correspondiente:




```
class Persona:
    def __init__(self, documento=0, nombre="No", apellido="No", edad=0):
        self.documento = documento
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad
```

Para instanciar objetos de esta clase Persona se debe ejecutar el constructor con una sintaxis que puede parecer extraña: en lugar de invocar en forma explícita a la función __init__, se indica el nombre de la clase (en este caso Persona) seguido de un par de paréntesis en el que se pueden enviar parámetros. Y hay más, ya que el primer parámetro formal (self) no debe ser enviado, el mismo va a llegar en forma automática al método constructor, referenciando al objeto que se está creando.

Los métodos cuyos nombres inician y terminan con dos guiones bajos (__) se denominan **métodos mágicos** ya que no son invocados en forma explícita con su nombres sino que se ejecutan en forma automática a partir de otras operaciones. En otro capítulo se los analiza con mayor detalle.

Por lo tanto, para obtener instancias de personas se las puede crear de la siguiente manera: requiere el siguiente bloque de código:

```
# Los atributos se inicializan con
# valores por defecto
a = Persona()

# Los atributose se inicializan con
# los valores indicados en los
# parámetros actuales
b = Persona(1234, "Juan", "Perez", 23)

# Los atributos documento y apellido
# se inicializan con los valores enviados
# y los otros con valores por defecto
c = Persona(documento=1234, apellido="Castro")
```

Las clases pueden estar programadas en cualquier archivo de código fuente y en cualquier parte del mismo (afuera de una función), sin embargo es usual ubicar cada clase en un archivo cuyo nombre coincida con el de la clase. De esta manera, la clase Persona podría estar programada en un archivo persona.py. Para utilizar la clase deberá ser importada, usualmente con una instrucción de la forma `from persona import *`.

7.5. Referencias

Las variables a las que se les asigna un objeto se crean con un tipo llamado object. Sin embargo, la idea de que la variable almacena un objeto, a pesar de ser cómoda, es incorrecta.

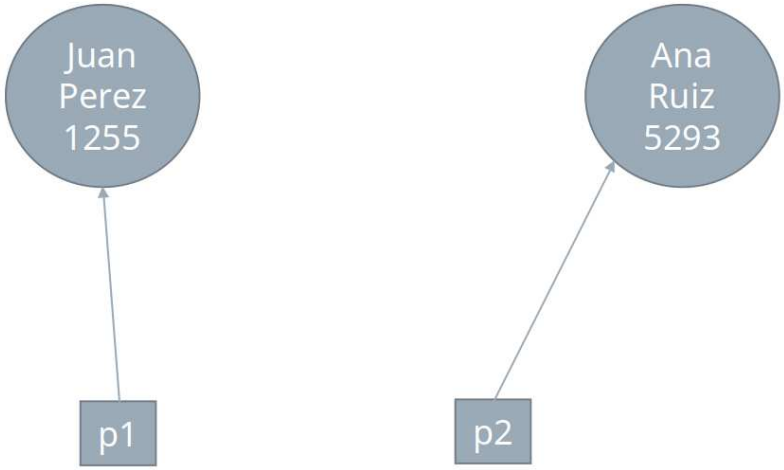
A diferencia de las variables de la programación estructurada, que son un almacenamiento que ocupa una porción de memoria con tamaño suficiente para que un dato entre completamente y que poseen un identificador, los objetos funcionan de forma diferente. La construcción de un objeto mediante una llamada a un método constructor efectivamente reserva memoria para almacenarlo, pero los objetos **no tienen nombre**!. Cada objeto posee una identidad provista durante su instanciación y que consiste en concreto en la dirección de memoria

donde esta almacenado. Pero sin un nombre no se les puede enviar mensajes y por lo tanto los objetos resultan inservibles.

Las variables de tipo object se denominan **referencias** y su valor efectivamente no es una instancia concreta de alguna clase. El dato que almacenan es la identidad de un objeto, es decir, la dirección de memoria donde el objeto se encuentra. Es práctico imaginar que una variable de tipo referencia guarda en realidad una flecha hasta el objeto.

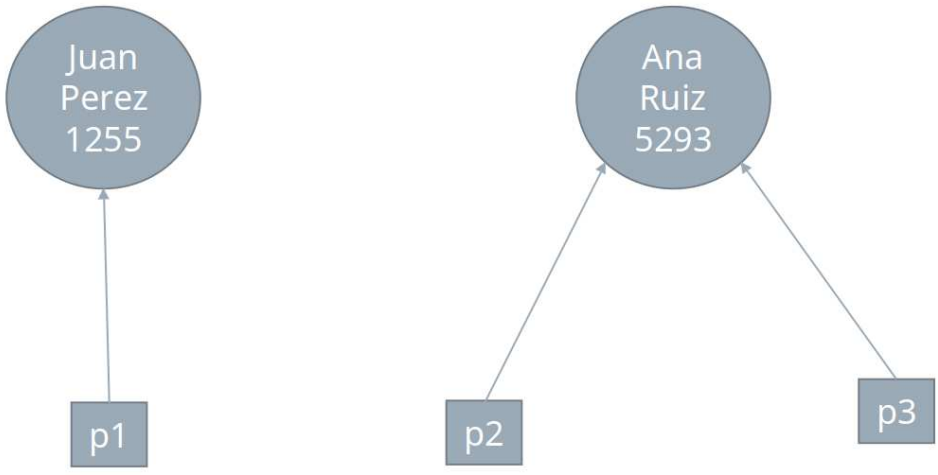
Por ejemplo, luego de ejecutar el siguiente código, los objetos quedan asignados como indica la figura:

```
p1 = Persona(documento=1255,nombre="Juan",apellido="Perez")
p2 = Persona(documento=5293,nombre="Ana",apellido="Ruiz")
```



Cada variable de tipo referencia puede señalar o apuntar a un único objeto, pero de la misma forma que dos variables pueden poseer el mismo valor, dos referencias pueden estar señalando al mismo objeto. En estos casos, el envío de mensajes desde una referencia o desde la otra afectan a la misma instancia concreta. La operación de asignación efectivamente copia los valores de las variables, y eso no es diferente en el caso de las referencias: si se asigna una referencia con el valor de otra, dado que ellas almacenan las “flechas”, ambas quedan referenciando al mismo objeto.

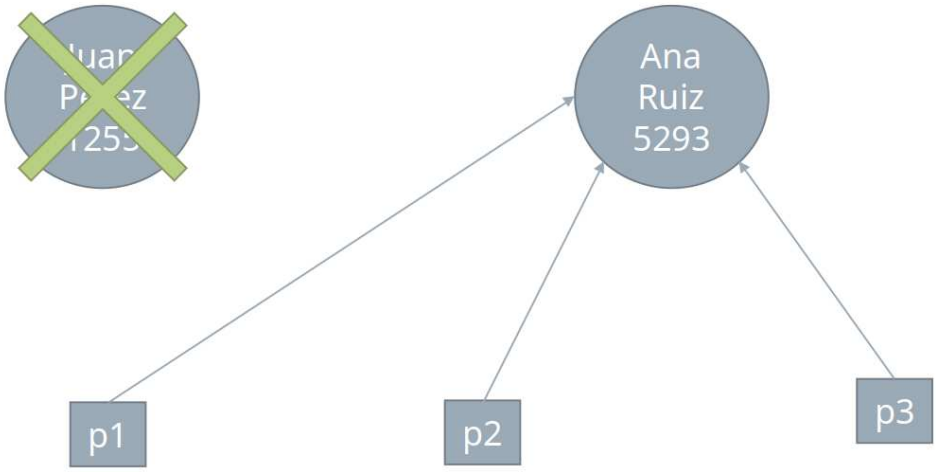
De esta manera, si a continuación del ejemplo anterior se ejecuta `p3 = p2`, el estado del programa queda como muestra la figura:



Otra consecuencia de esto es que la asignación no duplica objetos, la única forma de crear un objeto nuevo es mediante una llamada al constructor.

Finalmente, si un objeto creado deja de ser apuntado por alguna referencia, el mismo queda inaccesible y por lo tanto inservible, ya que no se le pueden enviar mensajes ni siquiera para consultar su estado. Eventualmente si hace falta memoria durante la ejecución del programa, el interprete de python podrá decidir eliminarlo de la memoria.

Si en el ejemplo se asigna `p1 = p2`, ninguna referencia queda apuntando a la persona Juan Perez:



7.6. Métodos

7.6.1. Introducción

La programación orientada a objetos propone la construcción de programas mediante la interacción entre objetos, los cuales son entidades que encapsulan tanto datos (atributos) como comportamientos (métodos) relacionados. En este enfoque, un programa se concibe como un conjunto de objetos que interactúan entre sí al intercambiar mensajes.

Cuando un objeto envía un mensaje a otro, está solicitando al objeto receptor que realice una acción o que proporcione información específica. Este proceso se denomina "paso de mensajes". El objeto receptor, a su vez, atiende el mensaje y ejecuta el método correspondiente para responder adecuadamente a la solicitud.

La forma de implementar este mecanismo de mensajes en los lenguajes orientados a objetos es a través de lo que se denomina "métodos". Los métodos son funciones asociadas a un objeto particular que definen su comportamiento y permiten que otros objetos interactúen con él. Los mismos pueden acceder a los atributos del objeto y realizar operaciones según la lógica definida en el código.

Los métodos que residen dentro de cada objeto y se ejecutan cuando el objeto recibe un mensaje. Los mensajes pueden contener parámetros que el método utilizará para llevar a cabo su operación, además de acceder a los atributos propios del objeto para actuar en consecuencia y enviar una respuesta como resultado del mensaje recibido.

En Python, los métodos se definen como funciones que reciben parámetros, pero el primer parámetro, por convención, es una referencia al propio objeto, que generalmente se nombra como `self`. Esta referencia self permite que el método acceda a los atributos y otros métodos del objeto actual, lo que facilita la manipulación de los datos y comportamientos específicos de ese objeto.

Además un objeto puede colaborar con otros objetos para llevar a cabo una tarea o cumplir con un objetivo específico. Cuando un objeto recibe un mensaje y atiende a dicho mensaje, puede requerir la colaboración de otros objetos para lograr su objetivo.

Para lograr esta colaboración, un método puede acceder a otros objetos y enviarles mensajes. Estos mensajes pueden incluir parámetros o información necesaria para que los objetos colaboradores realicen sus acciones. Luego, el método puede esperar las respuestas de estos objetos colaboradores y utilizar esa información para alcanzar sus propios objetivos y proporcionar una respuesta o resultado al mensaje original.

7.6.2. Sintaxis

Para agregar un método a una clase debe incluirse dentro del bloque `class` una definición de función con `def` cuyo primer parámetro sea `self`:

```
class Clase:

   def __init__(self, ... )
      # código del constructor

   def metodo(self, parámetros):
      # código del método
      return resultado
```

Así, en el contexto de una clase `Persona` con atributos `nombre` y `apellido`, es posible definir un método llamado `nombre_completo` que concatene ambos atributos y retorne el nombre completo de la persona. Este método se definiría dentro de la clase ”Personaz recibiría el parámetro ”self”, que es una referencia al objeto actual que recibe el mensaje de la siguiente manera:

```
class Persona:

    def nombre_completo(self):
        return f"{self.nombre} {self.apellido}"
```

Para enviar un mensaje de un objeto a otro, se utiliza la sintaxis de un punto seguido del nombre del método con paréntesis para enviar los parámetros necesarios. Esta sintaxis es equivalente a la de ejecutar una función, pero en este caso, está asociada a una referencia particular (objeto). En la lista de parámetros no se incluye al self, el mismo es llenado en forma implícita.

El operador punto es el que permite enviar el mensaje al objeto referenciado desde una variable de tipo referencia. Dado que las referencias pueden considerarse que almacenan una flecha que parte desde la variable y llega al objeto. Cuando se utiliza el operador punto en una referencia, este recorre la flecha hasta llegar al objeto al que está apuntando, y es a ese objeto al que se le envía el mensaje mediante la invocación del método.

```
per = Persona(123, "Juan", "Perez", 20)

print(per.nombre_completo())
```

Es importante tener en cuenta que si la referencia a la que se le envía el mensaje está vacía o tiene el valor `None` (nulo), se generará un error, ya que no hay un objeto apuntado al que pedirle nada. Por lo tanto, el programador debe garantizar que la referencia no esté vacía antes de enviar el mensaje.

Para evitar este tipo de errores, es común utilizar condicionales para verificar que la referencia no esté vacía antes de enviar el mensaje.

Capítulo 8

Encapsulamiento

8.1. Introducción

El encapsulamiento es un concepto fundamental en la programación orientada a objetos que se basa en la idea de que los objetos se comporten como “cápsulas”, es decir, como unidades cerradas que protegen su implementación interna. Esto significa que los detalles internos de la lógica, la naturaleza y la implementación de los atributos y métodos de un objeto están ocultos y no son accesibles directamente desde otros objetos. En lugar de permitir el acceso directo, los objetos deben comunicarse entre sí únicamente a través del paso de mensajes.

La finalidad del encapsulamiento es proteger y mantener la integridad y coherencia del objeto, evitando que otros objetos interfieran directamente con sus atributos o métodos internos. Esto promueve una mayor modularidad y flexibilidad en el diseño del software, ya que cada objeto puede cambiar su implementación interna sin afectar el funcionamiento de otros objetos que interactúan con él.

Al ocultar los detalles internos, el encapsulamiento mejora la abstracción y permite enfocarse en la interfaz pública del objeto, es decir, los métodos que están destinados a ser utilizados por otros objetos para interactuar con él. Esto crea una barrera protectora alrededor del objeto, lo que ayuda a prevenir errores y mantener la coherencia del sistema.

Para establecer una comunicación entre objetos, se utiliza el paso de mensajes, donde un objeto envía un mensaje a otro objeto solicitando la ejecución de un método específico o solicitando información. El objeto receptor responde a este mensaje atendiendo al método correspondiente y puede devolver el resultado o enviar un nuevo mensaje a otro objeto, lo que permite la colaboración y el trabajo conjunto entre los objetos del sistema.

8.2. Ventajas

8.2.1. Ocultamiento de la implementación

El encapsulamiento satisface la necesidad de permitir que un objeto pueda modificar su forma de funcionar sin afectar a los demás objetos con los que se comunica. Esta capacidad de adaptarse a cambios internos sin interrumpir la interacción con otros objetos es esencial para el mantenimiento y evolución del software a lo largo del tiempo.

Si se analiza por ejemplo un escenario donde un objeto tiene la responsabilidad de almacenar datos de manera persistente, podria hacerlo en un archivo de texto. Otros objetos interactúan con él enviándole mensajes para guardar o recuperar datos. En una primera versión del programa, esta puede ser la implementación utilizada, y los objetos que se comunican con él confían en que sus mensajes serán atendidos adecuadamente y que los datos se guardarán en un archivo.

Sin embargo, si en una versión futura el objeto decide cambiar su implementación para almacenar los datos en una base de datos o en una nube, en lugar de hacerlo en el archivo de texto, el objetivo es que este cambio no afecte al código existente de los otros objetos que interactúan con él.

Esto se logra a través del encapsulamiento. Los objetos que utilizan al objeto de almacenamiento no necesitan conocer los detalles internos de su implementación ni dónde se almacenan los datos. En su lugar, confían en la interfaz pública del objeto, es decir, los mensajes que pueden enviarle y las respuestas que pueden esperar.

De esta manera, los objetos que interactúan con el objeto de almacenamiento simplemente le envían mensajes como ”guardá este dato.º recuperá el dato almacenado”. No necesitan saber si el objeto utiliza un archivo de texto o una base de datos en la nube para almacenar los datos; simplemente confían en que sus mensajes serán procesados correctamente y que el objeto cumplirá con su responsabilidad de almacenamiento.

8.2.2. Validez del estado interno

El encapsulamiento surge también debido a la necesidad de mantener la validez del estado interno de un objeto. Un objeto requiere tener la tranquilidad de que sus atributos siempre poseen valores válidos acorde a su lógica interna. Si los atributos estuvieran expuestos y cualquier otro objeto pudiera modificarlos directamente, podrían asignarse valores que no cumplen con las restricciones lógicas del objeto. Como resultado, el objeto se vería forzado a incluir estructuras condicionales para verificar la validez de sus atributos cada vez que necesita operar con ellos. Esta situación generaría una complejidad innecesaria en el funcionamiento del objeto, requiriendo continuamente validaciones para garantizar la coherencia de los datos.

Entonces se recomienda ocultar los atributos de otros objetos y establecer que cualquier consulta o modificación de un atributo debe realizarse mediante el envío de mensajes específicos al objeto. De esta manera, el objeto tiene un control total sobre la validez de sus atributos y puede validarlos en el momento en que otro objeto intenta cambiarlos o asignarles un nuevo valor.

Por ejemplo, si un objeto representa un producto con un atributo de “precio”, que no puede ser negativo según la lógica del negocio, el encapsulamiento permite que cualquier intento de modificar el precio se realice mediante el envío de un mensaje específico al objeto, que incluya la nueva información. En este punto, el objeto puede verificar si el nuevo valor cumple con las reglas de validez o si es incorrecto. En caso de ser válido, el objeto procederá a asignar el nuevo valor y almacenar el dato correctamente. Por otro lado, si el valor no es válido, el objeto puede rechazar la asignación y tomar una decisión adecuada según el mecanismo que haya sido establecido, como lanzar una excepción, enviar un mensaje de error o tomar una acción alternativa.

Este enfoque permite al objeto garantizar que siempre tiene datos válidos y coherentes, ya que tiene el control total sobre la modificación de sus atributos. Asimismo, facilita el mantenimiento del código, ya que las validaciones y la lógica de negocio se concentran dentro del objeto, manteniendo una separación clara de responsabilidades entre objetos.

Se puede analizar esta característica a través de otro ejemplo. Cada persona tiene una billetera que contiene todos los billetes que posee en un momento dado. Cuando una persona necesita pedir prestado dinero a otra, debe enviarle un mensaje solicitando el préstamo. La persona receptora del mensaje conoce cuánto dinero tiene en su billetera y tiene el derecho de decidir si presta o no, ya que es su propio dinero. Puede responder afirmativamente, negativamente o incluso puede optar por no decir la verdad sobre la disponibilidad de dinero, aunque realmente lo tenga. Esta capacidad de decidir prestar o no es fundamental porque el dinero realmente le pertenece.

La importancia del encapsulamiento se hace evidente en este escenario. Si no existiera el encapsulamiento, es decir, si cualquier persona pudiera acceder directamente a la billetera de otra, podría tomar prestado dinero sin permiso y sin que el propietario se entere de la operación. Esto desencadenaría dos problemas. En primer lugar, el propietario de la billetera tendría menos dinero, lo que afectaría sus recursos.Pero el problema más significativo es que si el propietario no se diera cuenta de la reducción del dinero, podría tomar decisiones incorrectas al realizar compras o comprometerse con gastos sin saber que su dinero ha disminuido sin autorización.

Por lo tanto, para mantener la integridad de los datos y preservar la autonomía del objeto dueño de la billetera, se aplica el encapsulamiento. Los objetos ocultan la información sobre cuánto dinero tienen, de manera que no se pueda acceder directamente a su billetera desde otros objetos. La cantidad de dinero en la billetera solo será accesible si otro objeto envía un mensaje preguntando cuánto dinero tiene el propietario. Del mismo modo, si alguien solicita un préstamo, el objeto dueño del dinero decidirá si lo presta o no y nunca permitirá que su cantidad de dinero quede sin control.

8.2.3. Atributos calculados

Por otra parte, el encapsulamiento también brinda al objeto la capacidad de decidir si mostrar o no los datos almacenados en sus atributos cuando otros objetos los consultan. Esto permite que el objeto tenga el control sobre qué información se expone y cómo se presenta, según lo considere conveniente. Asimismo, el objeto puede optar por responder a las consultas calculando los datos a partir de otros atributos disponibles.

Cuando un objeto recibe un mensaje que consulta un dato específico, no es necesario que el objeto que envía el mensaje sepa si el dato que recibe es simplemente un atributo almacenado o si es calculado en el momento de la consulta. Este enfoque de abstracción oculta los detalles internos de la implementación y permite que el objeto proporcione la información solicitada de una manera coherente y transparente para el objeto solicitante.

Si por ejemplo un objeto “Círculo” tiene atributos como “radio” y “centro”, el mismo podría exponer métodos que simulen tener otros atributos, como “diámetro”, “superficie” y “circunferencia”. Estos atributos calculados podrían obtenerse en el momento de la consulta, basándose en los datos disponibles del círculo. Así una instancia de `Circulo` podría tener un método llamado `diametro()` que, en lugar de retornar el valor de un atributo, lo calcule

en función del radio disponible. Cuando otro objeto solicita el diámetro del círculo, simplemente accede a este método sin necesidad de saber si el diámetro es un atributo almacenado o si lo calcula en el momento.

8.3. Propiedades

Para implementar el encapsulamiento, una de las formas preferidas es utilizando el decorador `@property`. Este decorador permite programar los métodos de consulta y asignación, es decir, un método para obtener el valor de un atributo y otro método asociado para asignar un nuevo valor a ese mismo atributo. Cada uno de estos métodos, conocidos como getters (método de consulta) y setters (método de asignación), puede contener la lógica necesaria, tan sencilla o compleja como sea requerida.

Estos métodos se los decora especificando que conforman un par llamado **propiedad**. Una vez establecidos como propiedades, los objetos clientes pueden utilizar la misma sintaxis que utilizan para acceder a un atributo simple o para asignar un valor a un atributo. De esta manera, el objeto cliente sigue utilizando la sintaxis de atributos, pero en realidad, de forma implícita, se ejecutan los métodos de consulta y asignación correspondientes. Esto permite que el objeto reciba el mensaje de consultar o modificar un atributo sin necesidad de exponer directamente el acceso a sus atributos internos.

La sintaxis de estos métodos es bastante particular: el método de consulta se define con el nombre de la propiedad, generalmente un sustantivo, y se lo anota con @property. El método de asignación se anota con un decorador que lleva el nombre de la propiedad seguido de `.setter`, lo que indica que es el método setter asociado a esa propiedad. En el método setter, se recibe como parámetro self, como es habitual en todos los métodos, y el nuevo valor que se intenta asignar al atributo.

Por ejemplo, en la clase Persona del capítulo anterior, para que esté encapsulado el atributo `documento` se debería hacer:

```
class Persona:
    def __init__(self, documento, nombre, apellido, edad):
        self._documento = documento
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad


    @property
    def documento(self):
        return self._documento

    @documento.setter
    def documento(self, nuevo_documento):
        self._documento = nuevo_documento
```

Luego, cuando se intenta acceder al atributo documento en realidad se están ejecutando los métodos getter y setter:

```
p = Persona(11, "Juan", "Perez", 23)
p.documento = 22 #Ejecuta el método setter
print(p.documento) # Ejecuta el getter
```

8.4. Métodos mágicos

En Python, existen métodos que comúnmente se conocen como “métodos mágicos”. Aunque el término “mágicos”puede sonar confuso, estos métodos son efectivamente especiales y poseen características únicas que no se encuentran en otros lenguajes de programación, o que pueden variar de un lenguaje a otro.

Lo que distingue a estos métodos es que se invocan de manera implícita, es decir, no se llaman directamente utilizando la sintaxis tradicional de envío de mensajes o llamada a funciones. En lugar de eso, Python proporciona una forma especial de invocar estos métodos, mediante una convención de nomenclatura que los hace reconocibles. Estos métodos son conocidos como “Dunder methods”, una abreviatura de “double underscore methods”, debido a que sus nombres comienzan y terminan con dos guiones bajos (__).

Un ejemplo conocido de un método ”mágico.[es] `__init__`, que se utiliza como el constructor de una clase. Cuando se crea una instancia de una clase en Python, el método __init__ se llama de forma implícita para inicializar los

atributos y configurar el objeto. No es necesario invocarlo manualmente con su nombre ya que Python lo hace automáticamente al crear un objeto de la clase.

Un aspecto interesante de estos métodos mágicos es que permiten que las clases personalicen su comportamiento en ciertos contextos. Por ejemplo, el método `__str__` se utiliza para definir cómo se debe representar una instancia de una clase como una cadena de caracteres cuando se utiliza la función str(). Al definir este método en una clase, se puede controlar cómo se verá una instancia de esa clase cuando se imprima o se convierta en una cadena.

Estos métodos mágicos desempeñan un papel crucial en la programación orientada a objetos al proporcionar la capacidad de personalizar el comportamiento de las clases y los objetos creados a partir de ellas. Estos métodos son necesarios debido a la distinción fundamental entre los objetos creados a partir de una clase y los tipos de datos simples o primitivos que ofrece el lenguaje, como números, booleanos y cadenas.

En Python, los tipos de datos simples pueden participar en operaciones gracias a los operadores. Por ejemplo, los operadores de comparación permiten comparar dos números o dos cadenas, y las operaciones aritméticas aplicadas a números realizan cálculos aritméticos. Sin embargo, cuando se crean clases y se instancian objetos a partir de ellas, estos objetos no tienen la capacidad inherente de realizar operaciones entre ellos como la suma o la comparación de igualdad de sus valores.

Cuando se comparan por igualdad dos objetos, se obtiene simplemente un valor de verdad que indica si ambas referencias se refieren al mismo objeto o instancia. Sin embargo, no existe una forma natural de realizar comparaciones como “mayor que” o “menor que” entre objetos de una clase, ya que esta lógica no está definida de manera predeterminada para la mayoría de las clases personalizadas.

Con estos métodos se permite a los programadores agregar lógica específica a sus clases para habilitar operaciones personalizadas entre objetos de la misma clase. Por ejemplo, al definir los métodos `__lt__` (menor que) y `__gt__` (mayor que) en una clase, se puede personalizar cómo se deben comparar los objetos de esa clase en función de sus atributos internos. O, por ejemplo, con `__add__` para la suma, se consigue que los objetos realicen operaciones aritméticas personalizadas.

8.4.1. Método __str__

El método `__str__` es un componente fundamental en la programación orientada a objetos en Python. Este método se utiliza para proporcionar una representación en forma de cadena del estado interno de un objeto. Su lógica radica en mostrar, en formato de cadena, todos o algunos de los valores almacenados en los atributos del objeto. En otros lenguajes de programación, como Java o C#, este concepto se implementa mediante un método llamado toString, o el método asString de SmallTalk.

La ventaja de definir el método str con este nombre especial, con guiones bajos dobles (__str__), es que permite que Python realice conversiones implícitas del objeto a una cadena en diversas situaciones. Por ejemplo, cuando se utiliza la función print, esta puede recibir como parámetro una instancia de una clase definida por el usuario. En este caso, la función print invocará automáticamente al __str__ del objeto y mostrará por pantalla la representación que el objeto decide proporcionar de sí mismo en formato de cadena.

8.4.2. Métodos de comparación

La capacidad de comparar objetos de manera personalizada es esencial para la aplicación de lógicas específicas relacionadas con la igualdad, desigualdad y otros criterios de comparación.

Un ejemplo de estos métodos especiales es `__eq__`, que se utiliza para la comparación por igualdad. La lógica subyacente en este método es la de determinar si un objeto es igual a otro en función de los criterios definidos por el programador. Por ejemplo, en el contexto de una clase Persona, se lo podría programar para que devuelva True si todos los atributos de dos objetos Persona tienen el mismo valor. Esta lógica personalizada reflejaría la definición específica de igualdad para objetos de esta clase en el contexto de la aplicación.

Es importante enfatizar que Python no puede adivinar automáticamente cómo se deben comparar objetos personalizados. La razón detrás de esto es que la comparación entre objetos puede depender de lógicas de negocio específicas y variar ampliamente de una aplicación a otra. Por lo tanto los programadores deben proporcionar esta lógica de comparación personalizada en los métodos especiales correspondientes.

Además existen otros métodos especiales, como `__ne__` para la desigualdad, `__lt__` para “menor que”, `__gt__` para “mayor que”, `__le__` para “menor o igual que” y `__ge__` para “mayor o igual que”. Cada uno de estos métodos permite a los objetos de una clase personalizada participar en comparaciones utilizando los operadores correspondientes.

8.4.3. Métodos aritméticos

Considerando una clase que represente un concepto matemático, por ejemplo una Fracción, tiene sentido aplicar operaciones aritméticas sobre fracciones, tales como sumarlas o multiplicarlas. Un método especial que podría ser relevante en este contexto es `__add__`, que se utiliza para sobrecargar el operador de suma (+). Al programar este método dentro de la clase Fraccion, se le proporciona la capacidad de sumar dos instancias de fracciones. La lógica subyacente en el método determina cómo se realiza esta suma y qué significa en el contexto del dominio en el que se va a usar la clase.

A continuación se presenta una posible implementación de la clase Fraccion y de la sobrecarga de los operadores de adición y multiplicación:

```
from math import gcd

class Fraccion:
def __init__(self, numerador, denominador):
# Simplificamos la fracción al máximo común divisor
divisor_comun = gcd(numerador, denominador)
self.numerador = numerador // divisor_comun
self.denominador = denominador // divisor_comun

def __str__(self):
return f'{self.numerador}/{self.denominador}'

def __add__(self, otra_fraccion):
# Suma de fracciones: (a/b) + (c/d) = (ad + bc) / bd
nuevo_numerador = (self.numerador * otra_fraccion.denominador) + \
(otra_fraccion.numerador * self.denominador)
nuevo_denominador = self.denominador * otra_fraccion.denominador
return Fraccion(nuevo_numerador, nuevo_denominador)

def __mul__(self, otra_fraccion):
# Multiplicación de fracciones: (a/b) * (c/d) = (ac) / (bd)
nuevo_numerador = self.numerador * otra_fraccion.numerador
nuevo_denominador = self.denominador * otra_fraccion.denominador
return Fraccion(nuevo_numerador, nuevo_denominador)

# Crear dos objetos Fraccion
frac1 = Fraccion(1, 2)
frac2 = Fraccion(3, 4)

# Sumar dos fracciones usando el operador +
suma = frac1 + frac2
print(f'Suma: {frac1} + {frac2} = {suma}')

# Multiplicar dos fracciones usando el operador *
producto = frac1 * frac2
print(f'Multiplicación: {frac1} * {frac2} = {producto}')
```

En este ejemplo, se definió la clase Fraccion para representar números fraccionarios y se sobrecargaron los operadores + y * para que realicen la suma y multiplicación de fracciones según las reglas matemáticas correspondientes. La simplificación de fracciones se realiza en el constructor utilizando el máximo común divisor.

Con estos métodos, al crear dos objetos Fraccion (frac1 y frac2) se pueden usar los operadores + y * para realizar operaciones aritméticas con fracciones de manera más clara e intuitiva que invocando métodos con nombre.

Capítulo 9

Composición

9.1. Introducción

La composición es un principio fundamental que se utiliza para crear objetos complejos al combinar o componer a partir de objetos más simples. La composición es un tipo de relación entre clases en la que un objeto se compone de otros objetos como sus partes constituyentes.

La composición se basa en la idea de que un objeto puede tener referencias a otros objetos como sus atributos, y puede utilizar sus métodos para realizar acciones específicas.

9.2. Cardinalidad

Dependiendo de la cantidad de objetos involucrados en una relación de composición, la implementación puede variar sensiblemente. Si la cardinalidad es 0, 1 o alguna cantidad constante, puede implementarse mediante el uso de referencias simples, de forma tal que en la clase compuesta existan atributos cuyo tipo sea el de la clase contenida. Por ejemplo, en una clase que represente un partido de algún deporte de equipos, puede plantearse que un objeto de la clase Partido, estará compuesto de dos (y únicamente dos) instancias de la clase Equipo.

Cuando la cardinalidad indica que el objeto compuesto se relaciona con una cantidad grande, variable o desconocida de objetos de la clase relacionada, la implementación se orienta a disponer de una colección de objetos. Siguiendo con el ejemplo anterior, un equipo estará compuesto de muchos jugadores, por lo tanto la clase Equipo puede tener un atributo de tipo lista o diccionario que almacene instancias de la clase Jugador.

9.3. Implementación

Para implementar composición en Python se puede plantear un esquema similar al siguiente:

- En la clase Contenedora agregar un atributo de alguno de los tipos de colección.
- En el constructor de la clase contenedora no recibir un parámetro para asignar la colección, únicamente debe ser instanciada vacía.
- No agregar métodos de asignación y consulta para la colección.
- Agregar un método de agregado a la colección, que reciba como parámetro una instancia de la clase Contenida.
- Todas las responsabilidades que involucren a la colección, deben ser programadas como métodos en la clase Contenedora.

9.4. Ejemplo

Suponiendo una relación de composición entre las clases Biblioteca y Libro, estableciendo que una biblioteca está compuesta de 0 o más libros, se puede plantear la siguiente implementación:

```
class Libro:
def __init__(self, titulo, autor):
self.titulo = titulo
self.autor = autor

class Biblioteca:
def __init__(self):
```

```
    self.libros = []

    def agregar_libro(self, libro):
    self.libros.append(libro)

    def mostrar_libros(self):
    for libro in self.libros:
    print(f"Título: {libro.titulo}, Autor: {libro.autor}")

    # Crear libros
    libro1 = Libro("El Gran Gatsby", "F. Scott Fitzgerald")
    libro2 = Libro("Cien Años de Soledad", "Gabriel García Márquez")

    # Crear una biblioteca y agregar libros
    biblioteca = Biblioteca()
    biblioteca.agregar_libro(libro1)
    biblioteca.agregar_libro(libro2)

    # Mostrar los libros en la biblioteca
    biblioteca.mostrar_libros()
```